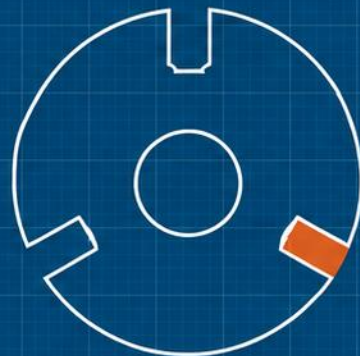
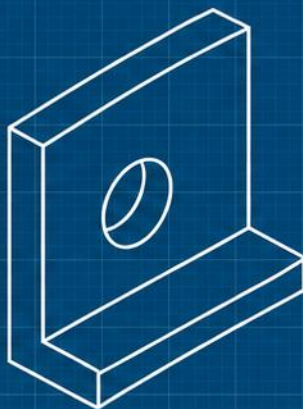
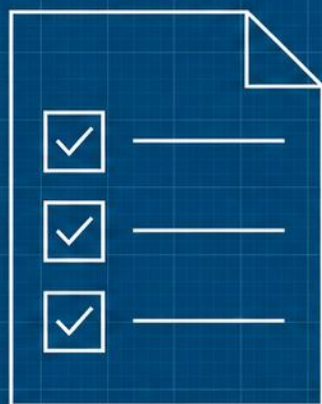


Bloc P00 · chapitre 7

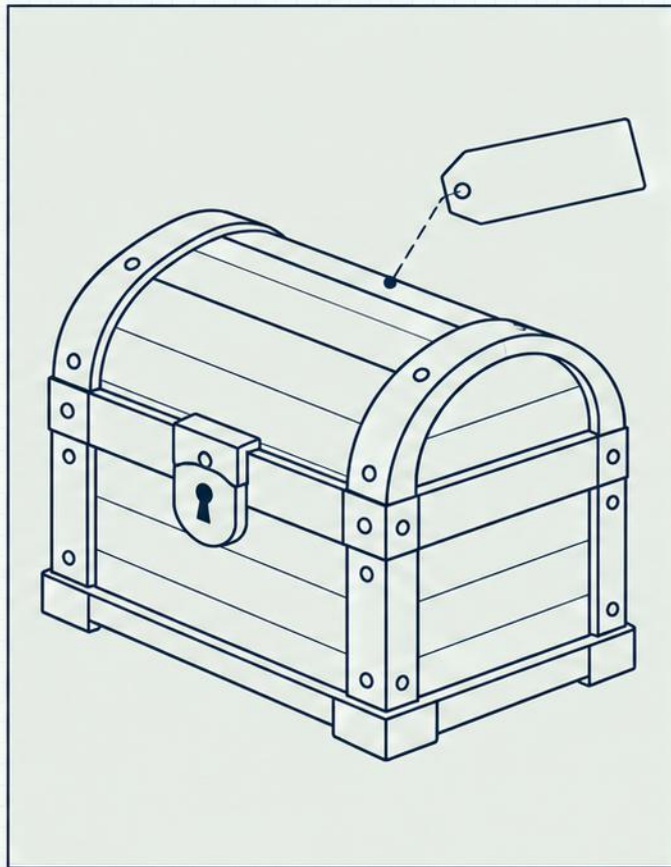
Contrats : interfaces, traits et enums

Dire ce qu'une chose sait faire, partager du code sans parent commun, n'autoriser que trois valeurs.



Pourquoi le trésor légendaire vaut-il 1 et pas 3 ?

```
1  final class Treasure
2  {
3      public function __construct(private string $rarity) {}
4
5      public function multiplier(): float
6      {
7          if ($this->rarity === 'rare') {
8              return 1.5;
9          }
10         if ($this->rarity === 'Legendary') {
11             return 3.0;
12         }
13
14         return 1.0;
15     }
16 }
17
18 echo (new Treasure('rare'))->multiplier(), PHP_EOL;
19 echo (new Treasure('legendary'))->multiplier(), PHP_EOL;
20
21
```



- Cinq minutes, en binôme. Ce code sent mauvais : un détail d'écriture y trahit un défaut de conception.

Un L majuscule, et aucun avertissement

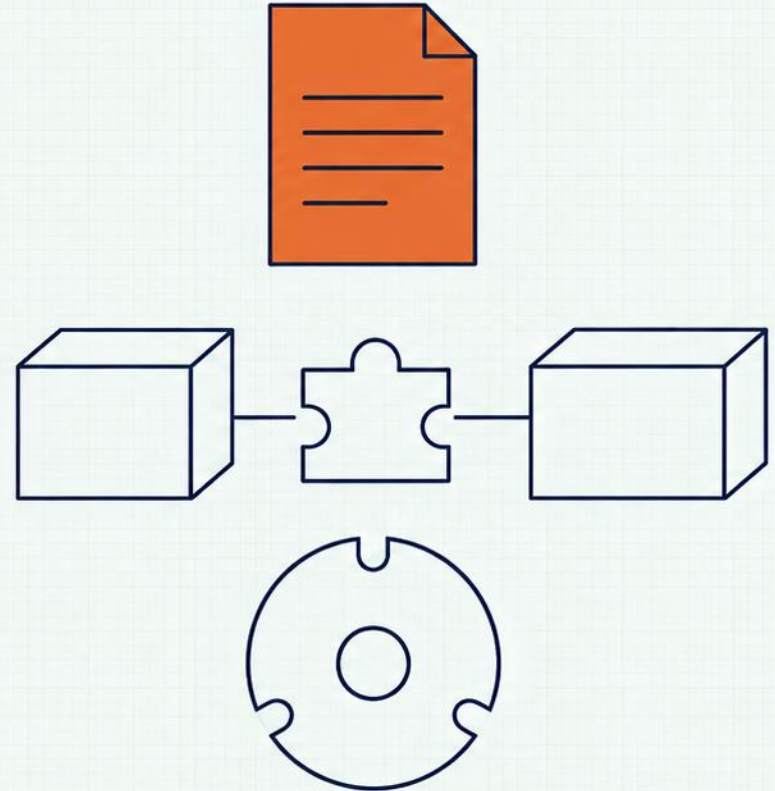
- 'legendary' n'est pas égal à 'Legendary' : le second if est faux et le code tombe dans le return 1.0, en silence.
- Le problème de fond : la rareté est un string, donc `new Treasure('bananes')` s'écrit sans erreur.
- La réponse du chapitre : une **enum**, un type qui n'accepte que trois valeurs.



Un seul parent ne suffit pas toujours

- Dire « ceci sait se battre » sans imposer une famille : **l'interface**.
- Partager du code entre deux classes sans parent commun : le **trait**.
- N'autoriser que trois valeurs, pas une de plus : **l'enum**.

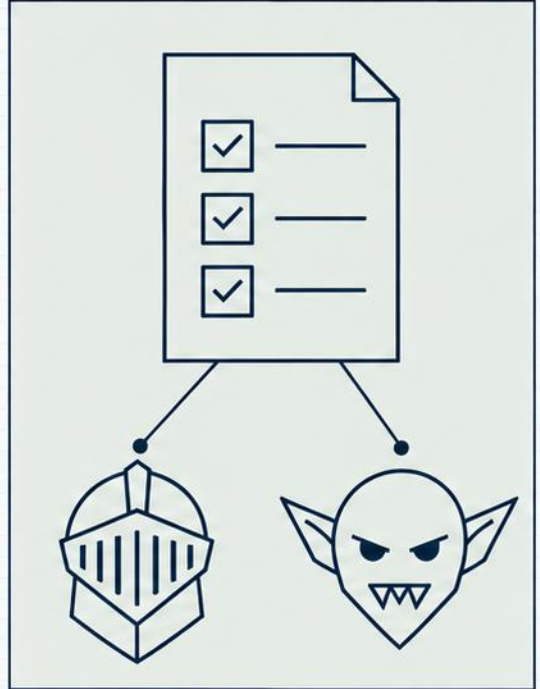
Vous les rencontrerez dans presque tous les fichiers de Laravel.



Une interface est un contrat : ce qu'il faut savoir faire, pas comment

```
1 interface Fighter
2 {
3     public function attack(): int;
4
5     public function takeDamage(int $amount): void;
6
7     public function isAlive(): bool;
8 }
```

```
1 final class Hero implements Fighter { /* les méthodes du chapitre 6 */ }
2
3 abstract class Monster implements Fighter { /* les méthodes du chapitre 6 */ }
```



- Que des signatures de méthodes publiques : le nom, les paramètres, le type de retour, et pas une ligne de code. Une classe déclare qu'elle remplit ce contrat avec `implements`, et PHP vérifie au chargement.

Une méthode manque : PHP refuse la classe au chargement

```
1  final class Ghost implements Fighter
2  {
3      public function attack(): int
4      {
5          return 1;
6      }
7  }
```

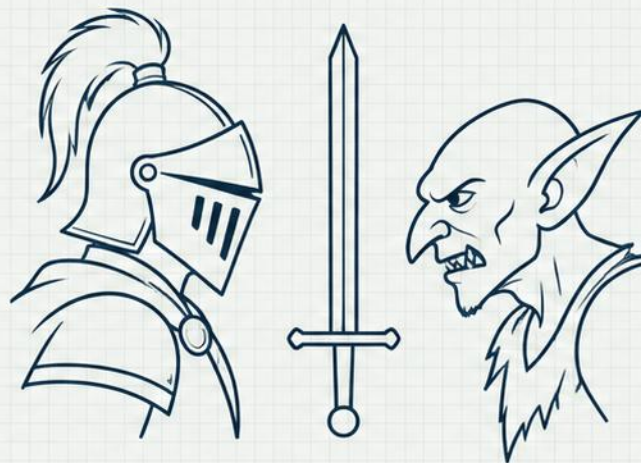
Fatal error: Class Dungeon\Ghost contains 2 abstract methods and must therefore be declared abstract or implement the remaining methods (Dungeon\Fighter::takeDamage, Dungeon\Fighter::isAlive)



Une classe concrète doit fournir les trois méthodes, directement ou par héritage.
Monster peut rester abstraite : ses classes filles remplissent le contrat pour elle.

Un paramètre se type par le contrat, pas par la classe

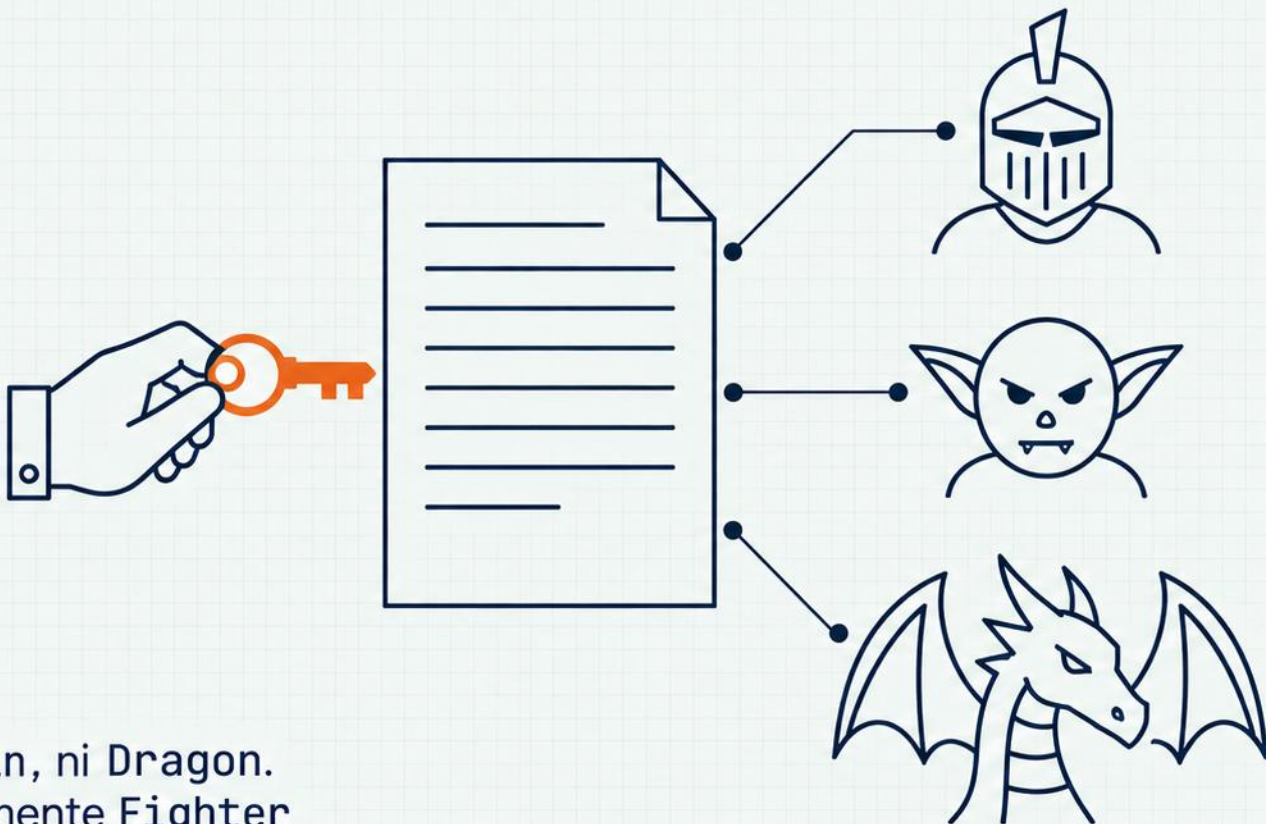
```
1  /** Ne demande ni un Hero ni un Goblin : deux objets qui savent se battre, c'est tout. */
2  function strike(Fighter $attacker, Fighter $target): void
3  {
4      $target->takeDamage($attacker->attack());
5  }
6
7  $arthur = new Hero('Arthur');
8  $goblin = new Goblin();
9
10 strike($arthur, $goblin);
11 strike($goblin, $arthur);
12
13 echo $arthur, PHP_EOL;
14 echo $goblin, PHP_EOL;
15 var_dump($arthur instanceof Fighter, $goblin instanceof Fighter);
16
```



```
Arthur (8/10 PV)
Gobelin (3/5 PV)
bool(true)
bool(true)
```

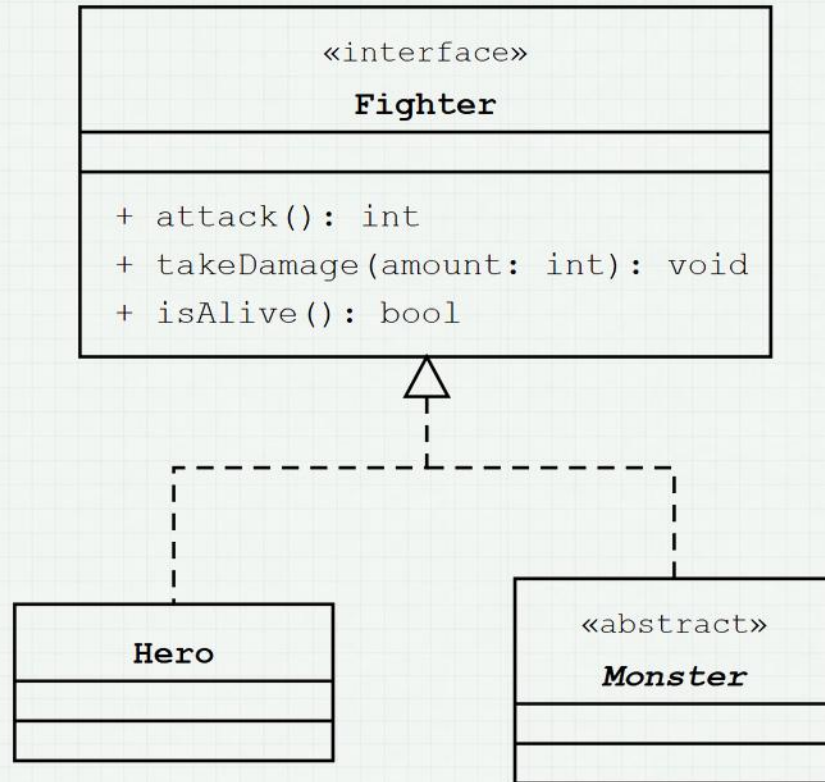
Un contrat sert d'abord à celui qui appelle

- `strike()` sait ce qu'elle peut demander à ses deux paramètres sans savoir de quelle classe ils sont.
- Des classes sans parent commun qui jouent le même rôle : Hero et Monster.
- Un objet qui doit se brancher sur une syntaxe de PHP, comme `count()`.
- Une pièce qu'on doit pouvoir remplacer par une autre, comme le dé truqué des tests.



`strike()` ne connaît ni Hero, ni Goblin, ni Dragon. Tout nouvel objet dont la classe implémente `Fighter` lui convient, sans modifier la fonction.

La réalisation : un trait pointillé terminé par un triangle vide, collé à l'interface

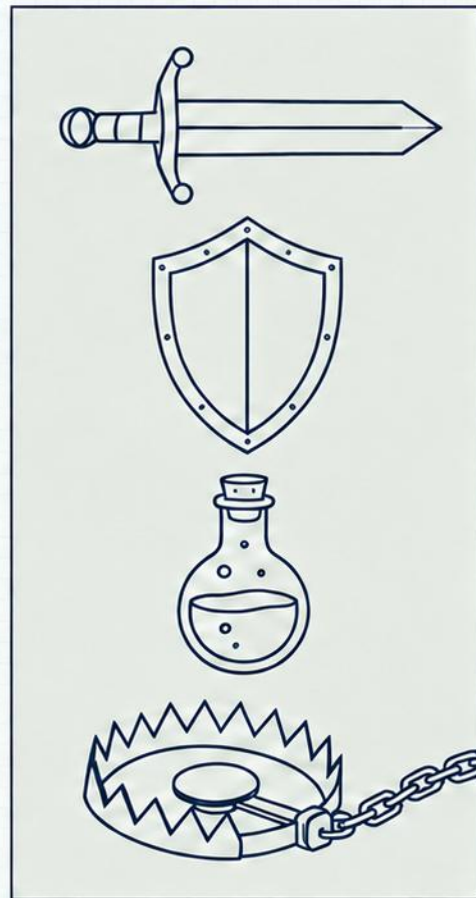


- Le trait se lit « réalise » : Hero et Monster remplissent le contrat Fighter, sans lien de parenté entre elles.
- La généralisation du chapitre 6 a le même triangle, mais un trait plein.
- Le mot-clé interface affiche «interface» au-dessus du nom.

PHP offrait une autre écriture : le type union Hero|Monster

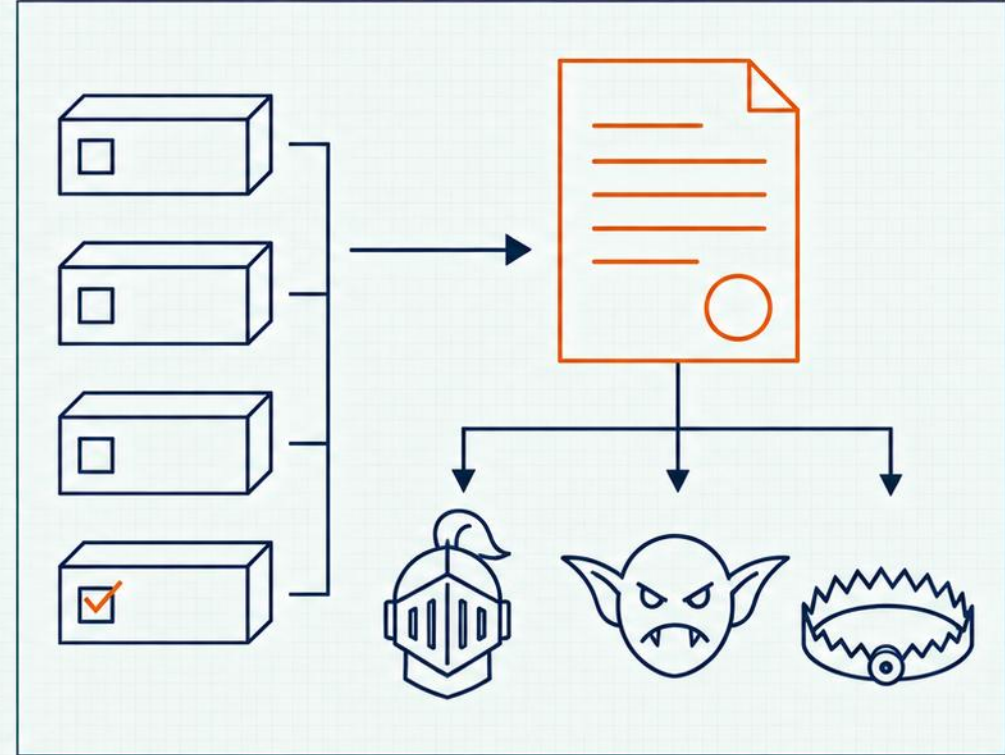
```
1  function strike(Hero|Monster $attacker, Hero|Monster $target): void
2  {
3      $target->takeDamage($attacker->attack());
4  }
```

- Un **type union** accepte l'un ou l'autre de plusieurs types, séparés par une barre verticale. Vous en croirez souvent : `int|float` pour un nombre, `string|null` pour une valeur qui peut manquer.
- Ici, l'écriture marche, mais elle liste des classes. Le jour où un Trap doit blesser le héros, il faut rouvrir `strike()` et allonger la liste.
- Rien ne garantit que `Hero` et `Monster` aient bien `attack()` et `takeDamage()` : PHP ne le vérifie qu'au moment de l'appel.



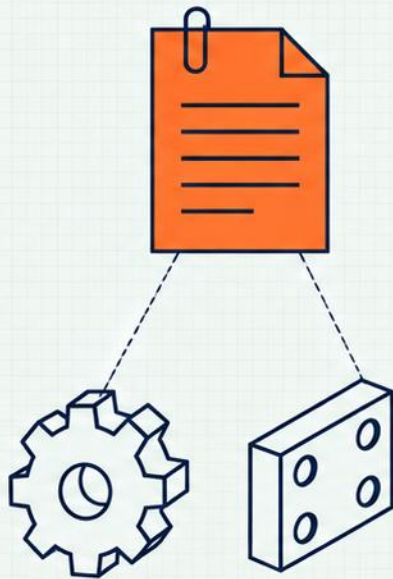
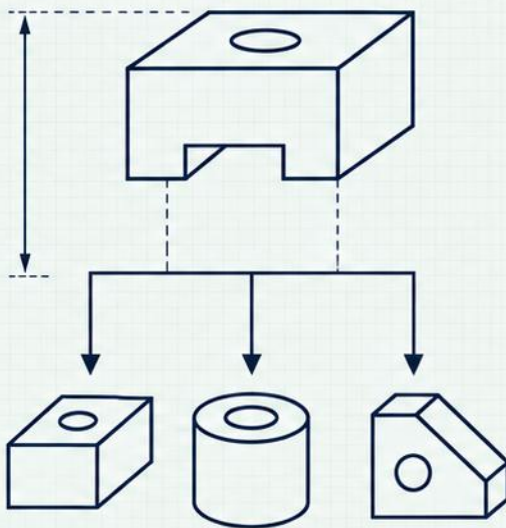
Le type union dit ce que l'objet est. L'interface dit ce qu'il sait faire.

- **Ce que le paramètre dit :** Hero | Monster, l'une de ces deux classes. Fighter, tout objet qui sait se battre.
- **Ajouter un Trap qui blesse :** rouvrir strike(). Ou écrire implements Fighter dans Trap.
- **Qui vérifie que attack() existe :** personne avant l'appel. Ou PHP, au chargement de la classe.
- **Quand l'utiliser :** deux types de nature différente, sans contrat commun, comme int | float. Ou plusieurs classes qui jouent le même rôle.



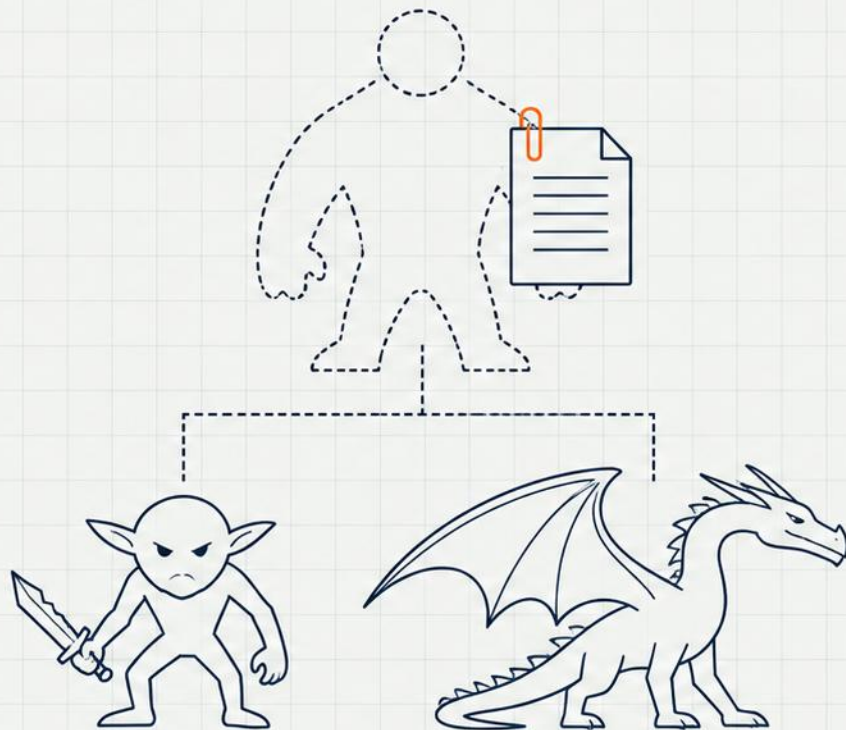
Même famille ? Classe abstraite. Même rôle ? Interface.

- **Peut contenir du code** : la classe abstraite oui, méthodes complètes et constructeur. L'interface non, que des signatures.
- **Combien par classe** : une seule classe abstraite, avec extends. Autant d'interfaces qu'on veut, avec implements.
- **Ce qu'elle dit** : « ceci **est un** ... » pour la classe abstraite. « ceci **sait faire** ... » pour l'interface.
- **Dans le Donjon** : Monster rassemble le code commun aux monstres. Fighter permet une même fonction de combat pour le héros et les monstres.



Une interface porte un rôle, une classe abstraite porte une famille

- Une interface porte souvent un rôle, comme `Fighter`, ou un nom en -able, comme `Countable` et `Stringable`.
- Une classe abstraite porte le nom d'une famille : `Item`, `Monster`, `Model`.
- Les deux vont souvent ensemble : `Monster` est abstraite **et** implémente `Fighter`.

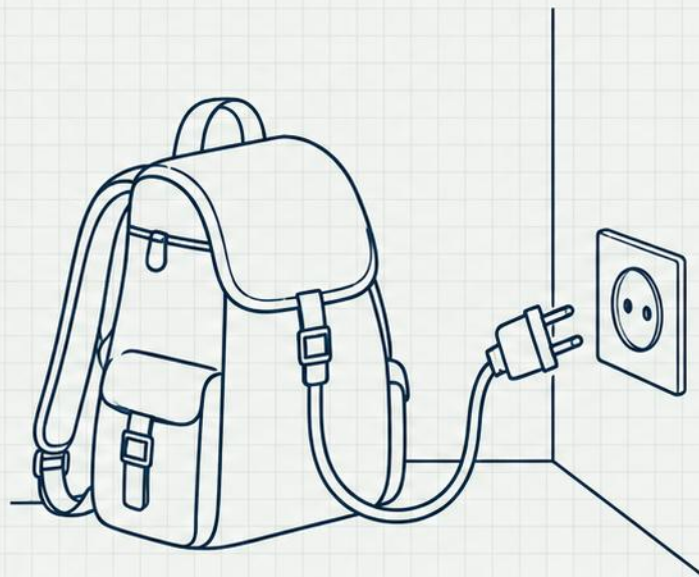


La fonction `count()` de PHP ne connaît pas votre méthode `count()`

```
1 $inventory = new Inventory();  
2 echo count($inventory);
```

Fatal error: Uncaught TypeError: count(): Argument #1 (\$value) must be of type Countable|array, Dungeon\Inventory given

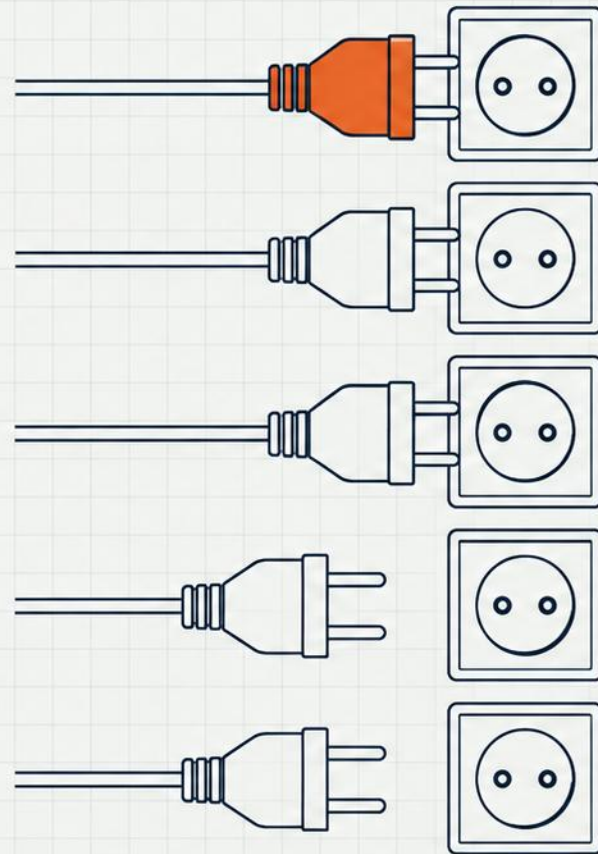
- L'Inventory du chapitre 6 a une méthode `count()` depuis le chapitre 5. La fonction du langage l'ignore.
- `foreach ($inventory as $item)` ne parcourt rien : PHP cherche des propriétés publiques, il n'y en a aucune, la boucle ne fait pas un tour et ne dit rien.



PHP fournit des contrats tout faits pour brancher vos objets sur sa syntaxe

- **Stringable** : écrire `__toString(): string` permet `echo $item;` et l'interpolation `"{$item}"`.
- **Countable** : écrire `count(): int` permet `count($inventory)`.
- **IteratorAggregate** : écrire `getIterator(): Traversable` permet `foreach ($inventory as $item)`.
- **JsonSerializable** : écrire `jsonSerialize(): mixed` permet de choisir ce que `json_encode()` produit.
- **ArrayAccess** : écrire `offsetGet(), offsetSet()...` permet `$inventory['épée']`.

Le Donjon utilise les trois premières. Les deux autres, vous les reconnaîtrez dans Laravel.



implements \Countable branche votre méthode sur la fonction du langage

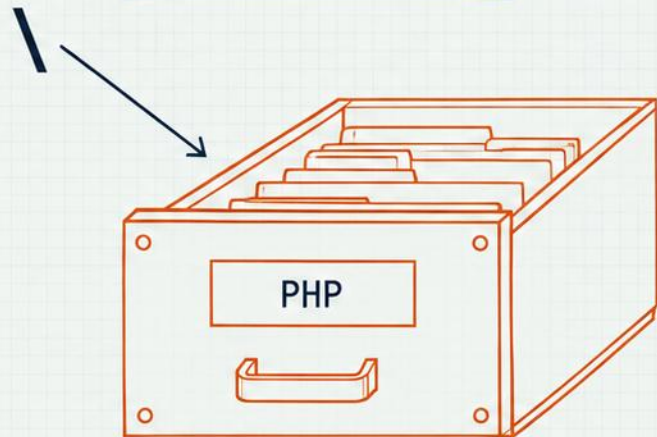
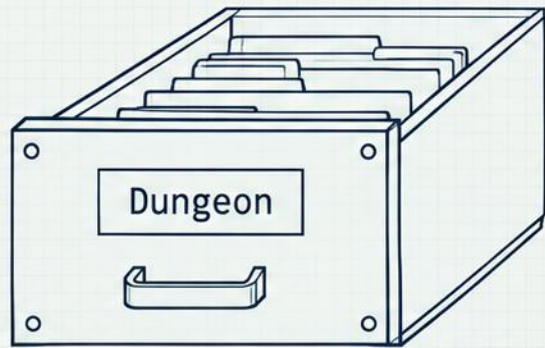
```
01  final class Inventory implements \Countable, \IteratorAggregate
02  {
03      /** @var Item[] */
04      private array $items = [];
05
06      /** Contrat de Countable : ce que count($inventory) doit renvoyer. */
07      public function count(): int
08      {
09          return count($this->items);
10      }
11
12      /** Contrat de IteratorAggregate : ce que foreach doit parcourir. */
13      public function getIterator(): \Traversable
14      {
15          return new \ArrayIterator($this->items);
16      }
17
18      // ... le constructeur, add() et totalWeight() des chapitres 5 et 6, has() et remove() du kata
19  }
20
```



La méthode count() existait déjà. Ce qui est neuf, c'est implements \Countable.

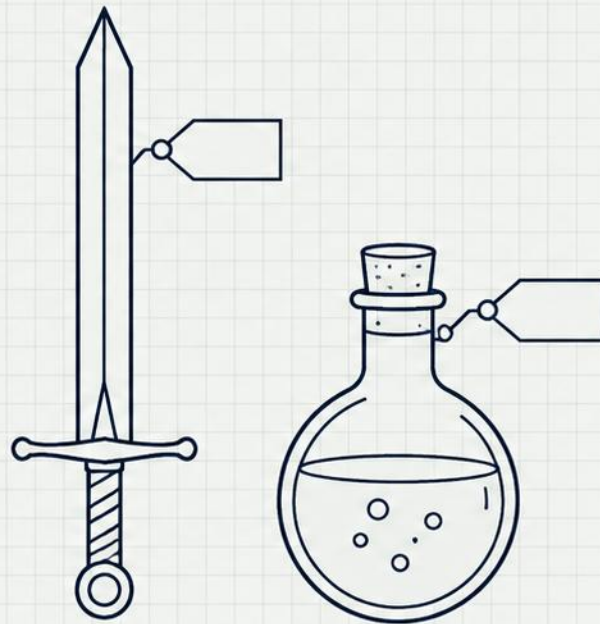
L'antislash : l'interface vient de PHP, pas du dossier Dungeon

- `\Countable`, `\IteratorAggregate`, `\Traversable` et `\ArrayIterator` viennent du namespace global de PHP, comme `\RuntimeException` au chapitre 6. Sans l'antislash, PHP chercherait `Dungeon\Countable` et ne le trouverait pas.
- `ArrayIterator` est une classe de PHP : donnez-lui un tableau, elle sait le parcourir. Avec `IteratorAggregate`, vous lui délégez le parcours.
- Avec `Iterator`, vous écririez vous-même les cinq méthodes du parcours : `current()`, `key()`, `next()`, `rewind()` et `valid()`.
- Le type de retour annoncé est `Traversable`, l'interface commune à tous les parcours.



Une classe qui déclare `__toString()` implémente `Stringable`, même sans l'écrire

```
1  abstract class Item implements \Stringable
2  {
3      // ... name, weight et describe() du chapitre 6
4
5      /** Contrat de Stringable : afficher un objet, c'est le décrire. */
6      public function __toString(): string
7      {
8          return $this->describe();
9      }
10 }
```



Depuis PHP 8.0, c'est automatique. L'écrire à la main documente le choix et permet de typer un paramètre par cette interface.

\$inventory n'est pas un tableau. Il se comporte comme un tableau grâce à deux interfaces.

```
1 $inventory = new Inventory();
2 $inventory->add(new Weapon('Épée courte', 2.0, 5));
3 $inventory->add(new Potion('Potion de soin', 0.5, 5));
4
5 echo count($inventory), ' objets', PHP_EOL;
6
7 foreach ($inventory as $item) {
8     echo '- ', $item, PHP_EOL;
9 }
```

2 objets

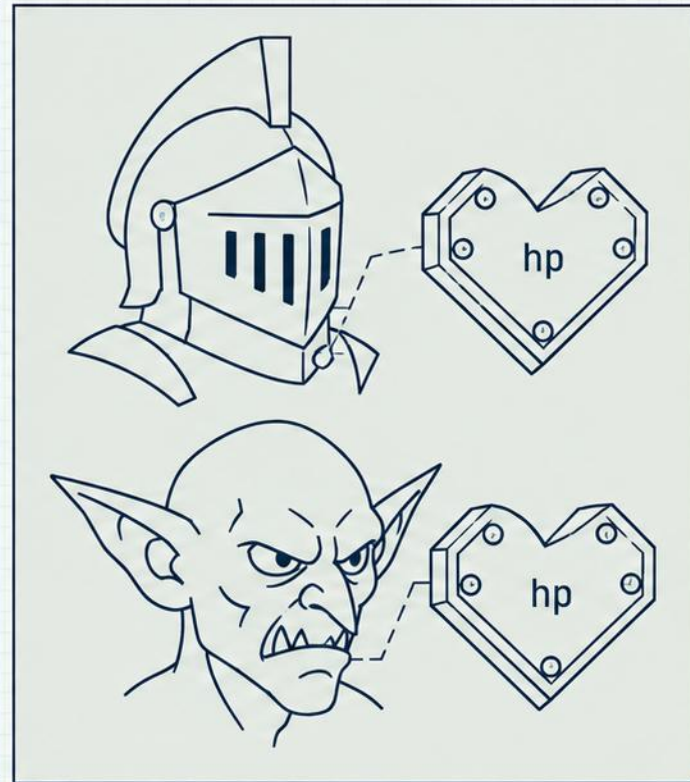
- Épée courte : arme (2 kg, 5 dégâts)
- Potion de soin : potion (0.5 kg, +5 PV)



- Le - de la deuxième ligne affiche \$item directement, grâce à Item::__toString().

Les mêmes méthodes de santé, mot pour mot, dans deux classes sans parent commun

```
1  final class Hero implements Fighter
2  {
3      public private(set) int $maxHp;
4
5      public private(set) int $hp {
6          set => max(0, min($this->maxHp, $value));
7      }
8
9      public function takeDamage(int $amount): void
10     {
11         // ... la vérification du chapitre 3
12         $this->hp -= $amount;
13     }
14
15     // ... et isFullHealth, heal(), isAlive()
16 }
```



- Monster contient les mêmes déclarations. Chaque règle de santé demande de modifier deux classes, et un jour l'une des deux sera oubliée. Elles ne peuvent pas hériter l'une de l'autre : un héros n'est pas un monstre.

Un trait : du code que PHP recopie dans la classe qui l'utilise, au chargement

```
1 trait HasHealth
2 {
3     /** Le maximum de points de vie. */
4     public private(set) int $maxHp;
5
6     /** Les points de vie courants : le hook les garde entre 0 et $maxHp. */
7     public private(set) int $hp {
8         set => max(0, min($this->maxHp, $value));
9     }
10
11     public bool $isFullHealth {
12         get => $this->hp === $this->maxHp;
13     }
14
15     public function takeDamage(int $amount): void
16     {
17         if ($amount < 0) {
18             throw new \InvalidArgumentException("Un dégât ne peut pas être négatif, $amount reçu.");
19         }
20
21         $this->hp -= $amount;
22     }
23
24     public function heal(int $amount): void
25     {
26         if ($amount < 0) {
27             throw new \InvalidArgumentException("Un soin ne peut pas être négatif, $amount reçu.");
28         }
29
30         $this->hp += $amount;
31     }
32
33     public function isAlive(): bool
34     {
35         return $this->hp > 0;
36     }
37 }
```

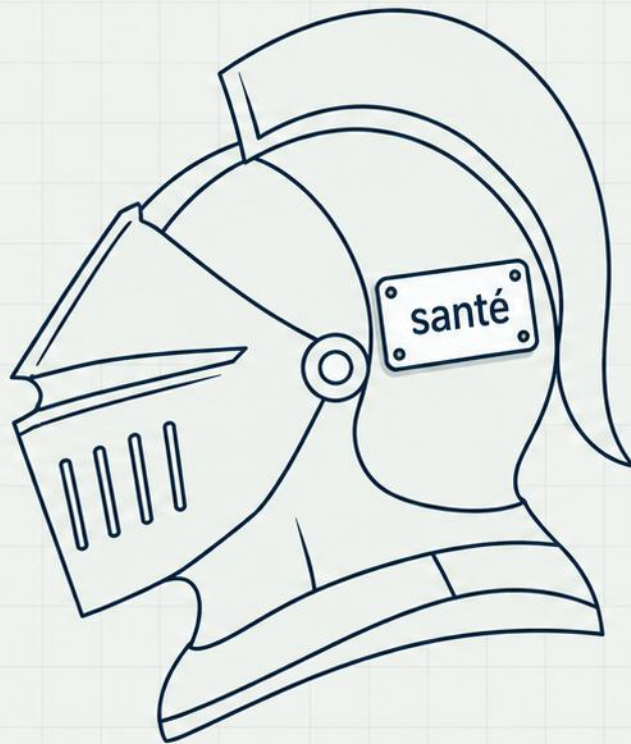


Il ne crée aucun lien d'héritage.

Une seule ligne dans chaque classe : use HasHealth;

```
1 final class Hero implements Fighter
2 {
3     use HasHealth;
4
5     public readonly Inventory $inventory;
6     public private(set) ?Weapon $weapon = null;
7
8     public function __construct(
9         public readonly string $name,
10         int $maxHp = 10,
11         public readonly int $strength = 2,
12     ) {
13         // ... les vérifications du nom et de maxHp, comme dans votre kata
14         $this->maxHp = $maxHp;
15         $this->hp = $maxHp;
16         $this->inventory = new Inventory();
17     }
18
19     public function attack(): int
20     {
21         return $this->strength + ($this->weapon?->damage ?? 0);
22     }
23
24     // ... equip(), drink(), __toString()
25 }
26
```

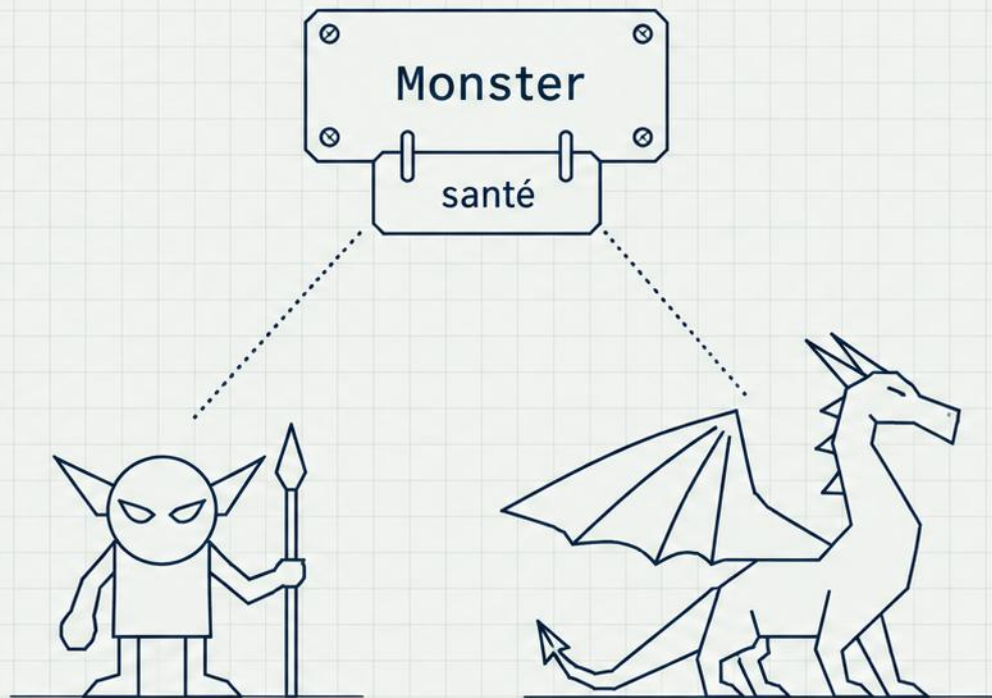
Les propriétés et les méthodes de santé ont disparu du fichier. Le constructeur n'a pas changé : il affecte toujours `$this->maxHp` et `$this->hp`, qui sont maintenant déclarées dans le trait.



Le contrat `Fighter` reste rempli : `takeDamage()` et `isAlive()` viennent du trait

`Monster` colle le même trait. `Goblin` et `Dragon`, ses classes filles, le reçoivent par héritage.

```
1  abstract class Monster implements Fighter
2  {
3      use HasHealth;
4
5      public function __construct(
6          public readonly string $name,
7          int $maxHp,
8      ) {
9          $this->maxHp = $maxHp;
10         $this->hp = $maxHp;
11     }
12
13     abstract public function attack(): int;
14
15     // ... __toString() du chapitre 6
16 }
```



Même mot, deux mécanismes : use importe en haut du fichier, use colle dans la classe

- **En haut du fichier**, après le namespace, use importe une classe : `use Dungeon\Dice;` du chapitre 2.
- **À l'intérieur d'une classe**, use colle un trait : `use HasHealth;`.
- **Quand vous lirez un fichier de Laravel**, regardez d'abord l'indentation.

```
use Dungeon\Dice;
```

```
{
```

```
    use HasHealth;
```

```
}
```

`class_uses()` liste les traits d'une classe, jamais ceux de ses parents

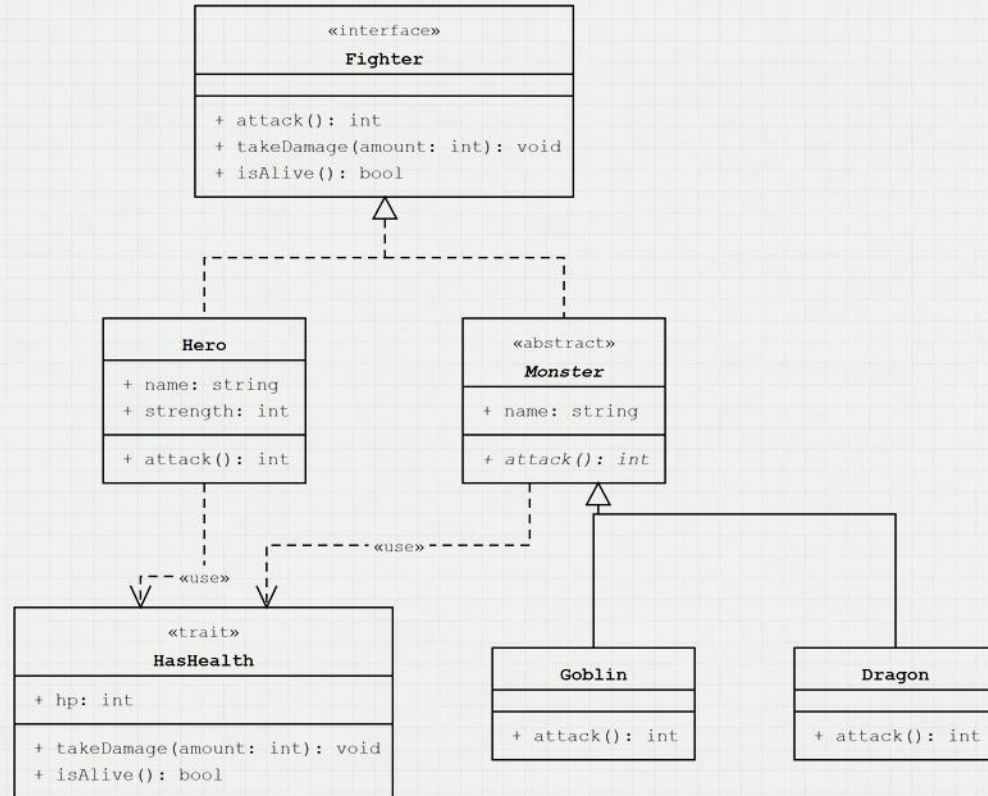
```
1 var_dump(class_uses(Hero::class));  
2 var_dump(class_uses(Goblin::class));
```

```
array(1) {  
    ["Dungeon\HasHealth"]=>  
        string(17) "Dungeon\HasHealth"  
}  
  
array(0) {  
}
```



Goblin utilise pourtant bien les méthodes de HasHealth, mais le trait est collé dans Monster, pas dans Goblin.

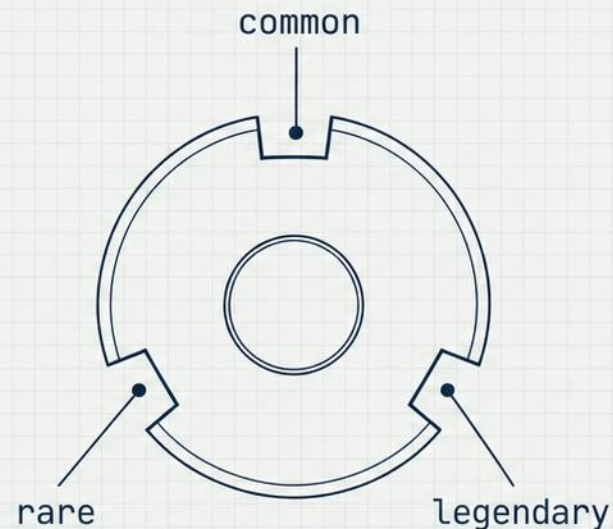
Trois flèches, trois idées : « est un », « sait faire », « se sert de »



- **Un trait n'est pas un parent.** Il permet de réutiliser du code. Une interface indique les méthodes attendues. Une classe abstraite peut faire les deux pour une même famille.
- UML n'a pas de trait : le cours le dessine avec un **stéréotype**. Le trait est une boîte de classe marquée «trait», avec ses propriétés et ses méthodes.
- Chaque classe qui le colle lui est reliée par une **dépendance** : un trait pointillé à pointe ouverte, étiqueté «use». Pas de triangle : ce n'est ni un héritage, ni un contrat.

Une enum : trois valeurs, pas une de plus, et des méthodes

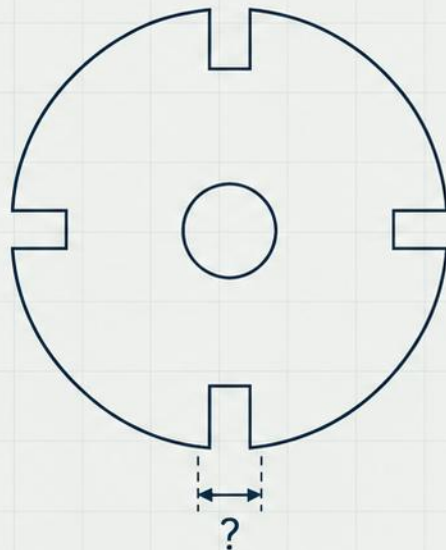
```
1  enum Rarity: string
2  {
3      case Common = 'common';
4      case Rare = 'rare';
5      case Legendary = 'legendary';
6
7      /** Combien vaut un objet de cette rareté, par rapport à un objet banal. */
8      public function multiplier(): float
9      {
10         return match ($this) {
11             Rarity::Common => 1.0,
12             Rarity::Rare => 1.5,
13             Rarity::Legendary => 3.0,
14         };
15     }
16
17     /** Le libellé affiché au joueur. */
18     public function label(): string
19     {
20         return match ($this) {
21             Rarity::Common => 'Commun',
22             Rarity::Rare => 'Rare',
23             Rarity::Legendary => 'Légendaire',
24         };
25     }
26 }
27 }
```



Avec : string après le nom, chaque cas porte une valeur texte : c'est elle qui s'enregistre en base de données ou en JSON. Une enum est une classe un peu particulière : ses cas sont des objets.

match compare strictement, renvoie une valeur et nomme le cas oublié

```
1  enum Rarity: string
2  {
3      case Common = 'common';
4      case Rare = 'rare';
5      case Legendary = 'legendary';
6      case Cursed = 'cursed';
7
8      // ... multiplier() et label() inchangés, sans ligne pour Cursed
9  }
10
11 echo Rarity::Cursed->multiplier();
```



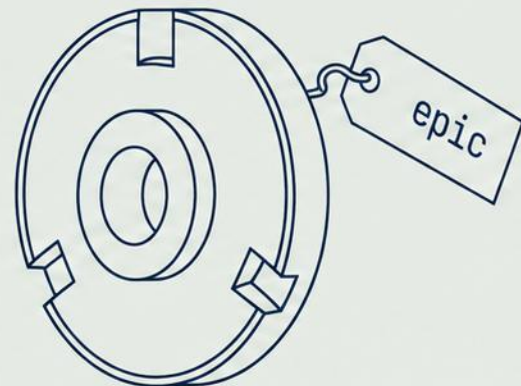
Fatal error: Uncaught UnhandledMatchError: Unhandled match case Dungeon\Rarity::Cursed

Comparez avec le `if` du début de chapitre, qui renvoyait `1.0` en silence. Ici, PHP s'arrête.

->name, ->value, cases(), from(), tryFrom()

```
1 $r = Rarity::Legendary;
2 echo $r->name, ' / ', $r->value, ' / ', $r->label(), ' / x', $r->multiplier(), PHP_EOL;
3
4 foreach (Rarity::cases() as $case) {
5     printf('%-10s %-10s x%.1f%s', $case->value, $case->label(), $case->multiplier(), PHP_EOL);
6 }
7
8 var_dump(Rarity::from('rare'));
9 var_dump(Rarity::tryFrom('epic'));
```

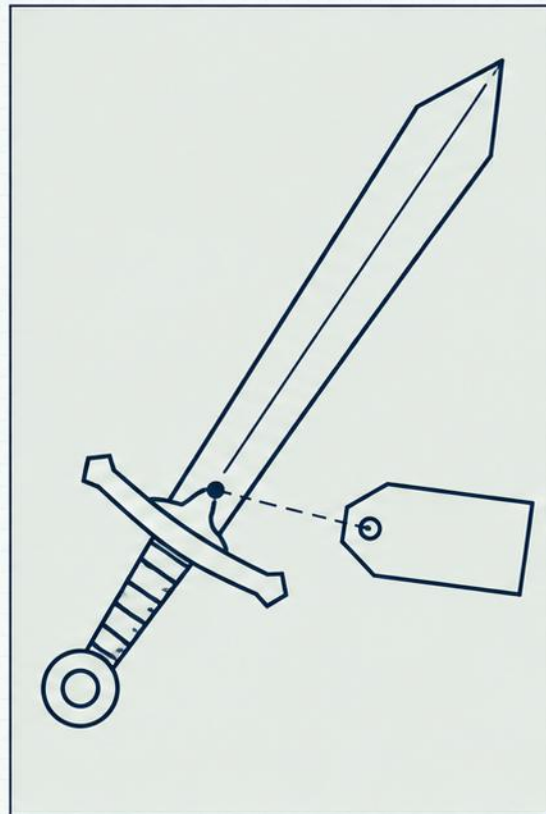
```
Legendary / legendary / Légendaire / x3
common      Commun      x1.0
rare        Rare        x1.5
legendary   Légendaire x3.0
enum(Dungeon\Rarity::Rare)
NULL
```



- ->name est le nom du cas, ->value sa valeur associée. cases() donne la liste complète, dans l'ordre de déclaration.
- from('epic') s'arrête sur une ValueError. tryFrom('epic') renvoie null : pour une donnée qui vient de l'extérieur.

Le paramètre est typé `Rarity` : une chaîne est refusée

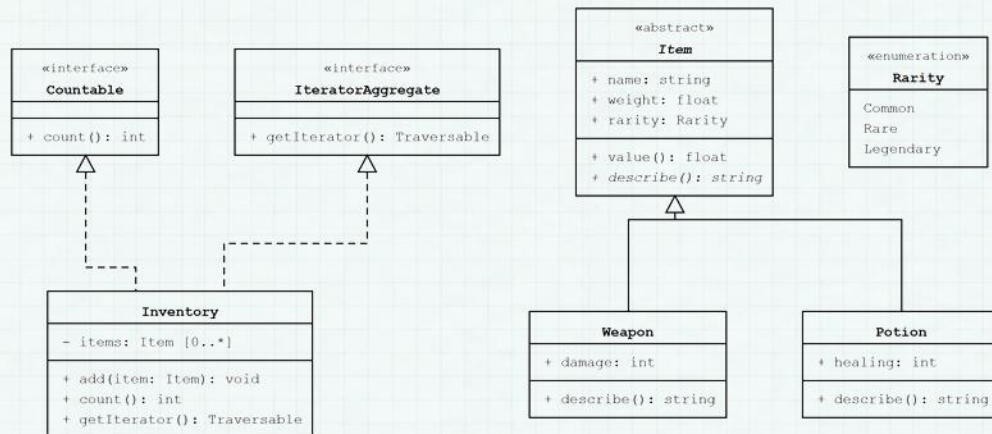
```
1  abstract class Item implements \Stringable
2  {
3      public function __construct(
4          public readonly string $name,
5          public readonly float $weight,
6          public readonly Rarity $rarity = Rarity::Common,
7      ) {
8          // ... la validation du poids du chapitre 5
9      }
10
11     /** Règle du jeu, arbitraire : la valeur marchande dépend du poids et de la rareté. */
12     public function value(): float
13     {
14         return $this->weight * $this->rarity->multiplier();
15     }
16
17     // ... describe() et __toString()
18 }
19
20 new Weapon('Épée courte', 2.0, 5, 'legendary');
```



Fatal error: Uncaught TypeError: Dungeon\Weapon::__construct(): Argument #4 (\$rarity) must be of type Dungeon\Rarity, string given

- **Weapon et Potion** passent la rareté au parent. Il faut écrire `Rarity::Legendary`, et l'éditeur propose les trois cas.

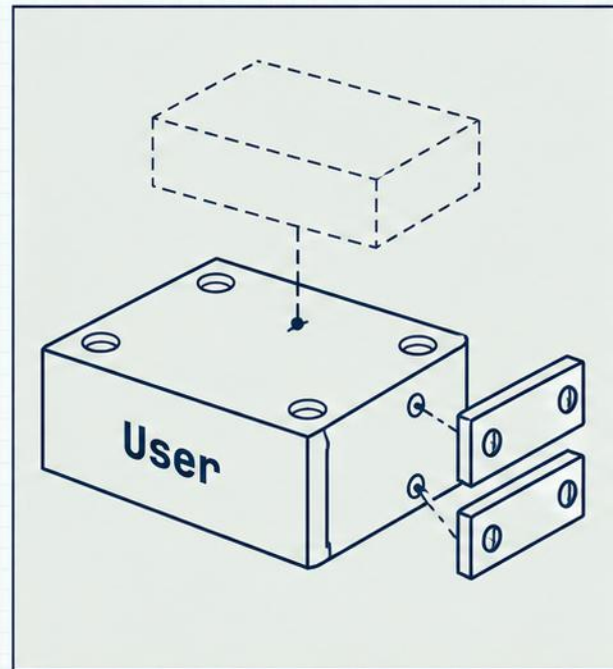
Une enum est un type de valeur : elle se lit dans la propriété, sans trait vers elle



- Le mot-clé `enum` affiche «enumeration», avec les cas dans le compartiment des attributs.
- Chaque objet porte une rareté : elle se lit dans `+rarity: Rarity`, comme `string` ou `int`. C'est la convention du chapitre 4 pour `Condition` dans `Copy`.
- Un inventaire **se compte** et **se parcourt** : deux réalisations vers les interfaces natives.
- Le losange vide qui relie l'inventaire aux objets est dessiné au chapitre 5 : ici, seule la propriété `-items: Item [0..*]` le rappelle.

Deux traits collés, un parent, une enum en base : le User de Laravel

```
1 use Illuminate\Database\Eloquent\Factories\HasFactory;
2 use Illuminate\Foundation\Auth\User as Authenticatable;
3 use Illuminate\Notifications\Notifiable;
4
5 class User extends Authenticatable
6 {
7     use HasFactory, Notifiable;
8
9     protected function casts(): array
10    {
11        return ['role' => UserRole::class];
12    }
13 }
```



- `use HasFactory, Notifiable;` : deux traits collés dans la classe. Aucun des deux ne dit ce qu'un `User` est. Ils ne peuvent pas être des parents : `User` en a déjà un.
- `extends Authenticatable` : l'héritage du chapitre 6. Les trois `use` du haut importent des noms.
- `'role' => UserRole::class` : en base, `'admin'`; en PHP, un cas de l'enum. Laravel fait le `from()` et le `->value` dans les deux sens. C'est Rarity, avec la base de données autour.

À vous : contrats, trait et enum dans le Donjon

Sur papier

1. Interface, classe abstraite ou trait ? Un Trap et une Potion qui savent tous deux **bless** un héros. Un Skeleton qui est un monstre. Un code de **journal des actions** que Hero, Monster et Room doivent tous porter.
2. `class_uses(Goblin::class)` rend un tableau vide alors que `$goblin->takeDamage(2)` fonctionne. Expliquez en deux phrases.
3. Écrivez l'enum Condition du chapitre 4, neuf, bon ou usé, avec une méthode `label()`, puis la propriété qui la porte dans Copy.

Kata niveau 4 : `composer test -- --group=niveau-4`

L'interface Fighter, les implements et les trois cas de Rarity sont fournis. Vous écrivez les corps des méthodes, le trait, et la rareté dans Item.

Bonus, niveau 5 : `composer test -- --group=niveau-5`

Battle fait s'affronter deux Fighter avec un dé reçu de l'extérieur, remplacé par FixedDice dans les tests.

